Interactive Ship Familiarization System: Technical Description*

Kenneth Wauchope
Navy Center for Applied Research in Artificial Intelligence
Naval Research Laboratory, Code 5512
Washington, DC

INTRODUCTION

Navy personnel first reporting on board ship must often spend a considerable amount of time learning their way around by asking for directions, receiving escorts, or just exploring. The use of computer-aided design in the development and construction of new ships provides a partial solution to this problem by making it possible to build interactive 3D virtual environments (VEs) that can be navigated and explored in advance of boarding the actual ship. Such environments, however, must be augmented with intelligent query answering and route finding capability to be fully useful. This in turn requires that the purely navigation-oriented VE interface be integrated with a symbol-based interface that can provide information such as compartment names and numbers. While such an interface would typically be implemented as a GUI (Graphical User Interface) in most computer applications, speech interaction provides a highly natural alternative that offers minimal interference with the eyes/hands-busy task of virtual navigation. We have named the resulting hybrid technology Augmented VE (AVE) since it is an exact parallel to so-called Augmented Reality (AR) technology in which an information system is overlaid on the (real) 3D world via a device such as a see-through head-mounted display. A speech-interactive VE system is then an AVE in which speech is a primary available interface medium.

This paper gives a technical description of the Interactive Ship Familiarization System (ISFS), a speech-interactive VE system for ship familiarization which is a transition from an earlier system called the Multimodal Ship Familiarization Tool (MSFT) [Tate et al. 2000] to an new application domain and software environment. The MSFT domain is a 3D model of a significant portion of the Navy's Ex-USS *Shadwell* fire research and test ship at Mobile, Alabama. The new domain is a model of the Island House structure (Levels 08 & 09) of the USS *Ronald Reagan* (CVN76) aircraft carrier, and includes not only ship structure but also furnishings, electronic instrumentation, utility piping and ductwork, fire extinguishers and other machinery, all modeled in accurate detail. The MSFT software was written using the World Toolkit, a C-language based SDK (System Development Kit) from Engineering Animation, Inc., and runs on an Intergraph 500MHz Pentium III computer. ISFS is written in VRML (Virtual Reality Modeling Language) using Java and ECMAScript as scripting languages, and currently runs on a Dell 2.0GHz Xeon machine in a standalone VRML viewer developed using the Cortona VRML Client SDK from ParallelGraphics Inc.

VRML is a scriptable, text-based 3D modeling language that runs on a variety of hardware platforms and software environments, and has been widely used for industrial and scientific applications by numerous academic and US government agencies such as NCSA, NIST, NASA, and NSF. While it may soon be superseded by its successor X3D (a member of the XML extensible markup language family) and other support technologies like Java3D, reverse

* AIC Technical Report AIC-03-001, Navy Center for Applied Research in Artificial Intelligence, Naval Research Laboratory, Code 5512, 4555 Overlook Ave. SW, Washington DC 20375-5337, March 2003.

translators from VRML to X3D should enable past and current VRML development work to remain viable into the future.

PILOT PROJECT: ITD KIOSK

The development of the ISFS was facilitated by a pilot project, the Information Technology Division (ITD) Multimodal Information Kiosk, in which we first explored and developed techniques for implementing speech- and GUI-controlled route finding and animated escort capabilities in VRML and Java. The ITD Kiosk currently models two of the division's main office buildings at NRL and provides both an office directory graphical interface and speech command interface for requesting virtual escorts to the offices of ITD personnel. The model includes interactive animated elevators, stairways, realistic lighting effects, texture maps, sound effects, and use of photographic imagery to enhance realism of exterior views and distinctive furnishings.

ISLAND HOUSE 3D MODEL

The 3D model of the CVN76 Island House Levels 08 & 09 was provided to NRL by the Newport News, VA facility of Northrop Grumman. The model was originally developed in the FTL file format on a proprietary 3D modeling software called Vivid, and was converted to VRML for delivery to NRL. Initially the model was delivered as a single 2GB VRML file which (on our 512MB machine) was too large either to run in a VRML viewer or to edit in our model development software, Autodesk's 3D Studio VIZ 3i. Subsequently the model was redelivered broken down into nine smaller files each containing a different layer of the original, as follows:

Waveguide	765MB
Electrical Equipment	723MB
Piping	334MB
Structure	322MB
Wireways	222MB
Hull Outfit	169MB
Machinery	129MB
Ventilation	69MB
HVAC Insulation	8MB

These VRML files were now of an appropriate size to import into 3D Studio, edit in its native format, and re-export as new VRML files acceptable both to the VRML viewer and to several postproduction development tools for data and geometry complexity (polygon) reduction.

From the standpoint of creating a recognizable and navigable space containing a fairly realistic clutter level of furnishings and equipment, the essential files for our purposes were Structure (floors, walls, ceilings, doors and hatchways), followed by Hull Outfit (chairs and other furniture, stairs, railings, ladders, hatchway doors, electrical equipment mounts, binoculars) and finally Electrical Equipment (computer monitors and keyboards, other device consoles, lights, electronics cabinets). While initially the Structure file was one of the largest (despite modeling features like floors and walls that should only require relatively few simple polygons), we discovered experimentally that by converting the original VRML file to AutoCAD DXF format and then re-exporting the file as VRML we were able to reduce its size by a factor of 40, from 322MB to only 8MB, with no loss of detail and only a few minor disruptions in geometry. (VRML files generated by 3D modeling programs have a reputation for being bloated in size, and apparently this was one such case.) Data reduction (redundancy elimination) on the remaining

files typically decreased their size by a factor of 5-6 or so. Since the Electrical Equipment file was by far the largest we were using, it was additionally submitted to 50% polygon reduction without showing any significant deterioration in visual appearance.

After postproduction we were able to load the Structure and Hull Outfit files for both levels and the Electrical Equipment for one level (09) without exceeding the 320MB or so of RAM available to the VRML viewer, thus keeping the computer's total memory usage within its 512MB RAM limits and avoiding the virtual memory paging that otherwise would interrupt the walkthrough animations with long pauses and leaps ahead in time. Since that particular computer can be upgraded to as much as 2GB RAM, more if not all the Island House geometry could potentially be loaded, although it is uncertain whether the resulting animation frame rate would remain at the current minimally acceptable level of about five frames per second. One VRML optimization we experimented with was to set the visibilityLimit so that geometry further away than, say, fifteen meters is not displayed at all. So long as no line of sight is longer than that the technique can be effective, but with the speedups provided by file cleaning and polygon reduction we felt we no longer needed that slight increase in efficiency. Another optimization (used in the ITD Kiosk) was to monitor the user's position with a ProximitySensor and switch relevant geometry in and out using a VRML Switch node. That is practicable in situations when the switched geometry lies completely outside the user's field of view (e.g. when the user is in a closed elevator), but is difficult to implement cleanly when there is visual overlap between the switched geometries, as when approaching doorways, climbing stairs, etc. In any event, our experiments with geometry switching in the current project had no effect on frame rate, so the geometry may already be grouped in a browser-efficient manner to begin with.

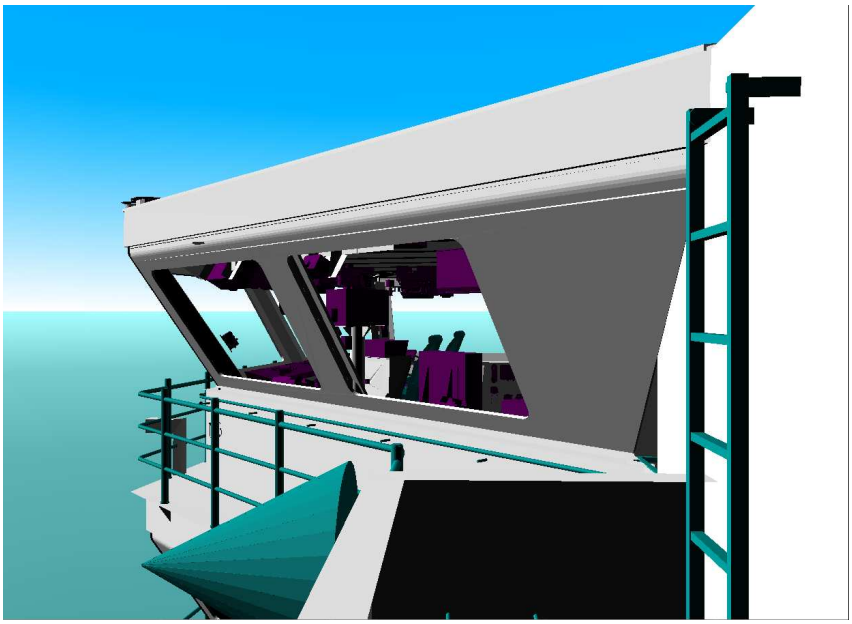

Fig. 1: View forward from 09 Level port weather deck, virtual *Reagan*

While detailed in geometric appearance, the modeled objects are not realistically colored but rather are color-coded by layer (Structure: gray, Hull Outfit: cyan, Electrical Equipment: magenta, etc.) making for a somewhat surreal virtual environment (Figure 2). While it would be a simple matter to change the color for an entire layer in the model editor, the absence of object types makes it impossible to, say, select all fire extinguishers to assign a particular color, so colorization would instead have to be done laboriously on an object-by-object basis. Similarly,

attempting to optimize the model by using the editor to delete individual bits of geometry considered non-essential to the VR display would be prohibitively time-consuming and would only end up "nickel-and-diming" the overall polygon count by a few tenths of a percent at most. The original model is lit by three DirectionalLights, which illuminate all geometry of the same orientation equally throughout the world. While inexpensive for the VR browser to render, such lights produce an artificial "glare" that is not well suited for realistic interior scenes but is entirely consistent with the otherworldly color scheme that currently exists. (The ITD Kiosk achieved much of its realism through the use of dozens of individual PointLights that create pools of light that attenuate with distance, but it could afford their increased rendering costs because of its considerably smaller polygon count.) We also added a Background node providing sky, sea and horizon since otherwise the Island House would appear to be floating in the blackness of space, and that addition did much to improve the believability of the scene and ameliorate any perceived harshness in the lighting and colors.

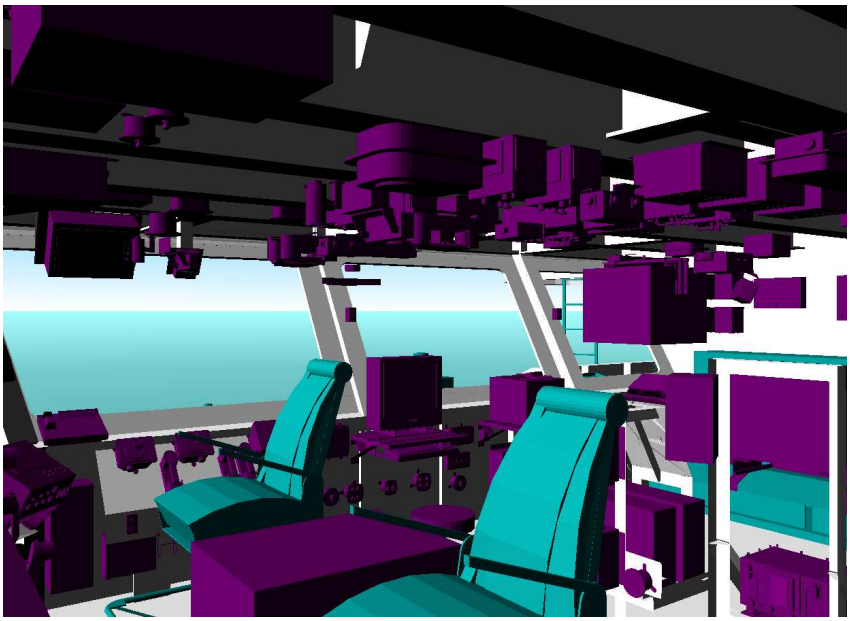

Fig. 2: Fly Control, virtual Reagan

While the most familiar VRML viewers are Web browser plug-ins, that runtime environment imposes security limitations on what system operations the embedded Java is permitted to execute. In our case, Java was unable to establish a connection with the speech recognition engine because browser security was enforcing restrictions on the opening of arbitrary system resources. This is understandable since one would certainly not like to be browsing a remote VRML file on the Web that has the ability to open a microphone connected to your machine and start listening in on your office conversations. While it is possible for Java to interactively request security override permissions from the user, different vendors and releases of Java handle this in substantially different ways, including the older version of Java (Microsoft JVM 1.1) that the Cortona VRML viewer uses. One workaround (used in the ITD Kiosk) was to have the VRML program open a TCP/IP socket connection to an external speech I/O agent, an operation that also requires a security override but one which we knew how to implement in an old release (4.0) of one particular Web browser (Netscape). However we finally decided that the simplest solution would be to run the VRML program in a stand-alone viewer built using the Cortona SDK, thus bypassing browser security issues completely. This approach had the further advantage of allowing us to customize the interface to the VRML viewer itself if we so desire.

The original VRML files from Grumman also made use of VRML's Level Of Detail (LOD) feature in which simpler geometry is used for distant objects and a more detailed geometry is switched in as the object reaches a certain range from the viewer. However the range figure provided in the original VRML files was too small by a factor of ten for the Cortona viewer, only swapping in the detailed geometry when less than a meter away. Scaling up the LOD range solved this problem, but ceased to be an issue when we began importing the VRML files into 3D Studio for editing since that program only retained the close-range geometries and apparently discarded the others. This actually turned out to be beneficial, because the original Structure file with LOD commanded a 50-70% CPU load even while displaying a static scene, whereas with 3D Studio's LOD-free file the CPU load dropped to 0%. Importation into 3D Studio also disabled the surface smoothing present in the original files, which we were unable to restore by experimenting with alternative crease angle values or use of surface normals. We also found it critical that VRML be exported from 3D Studio with six digit precision to retain the quality of the original geometry.

PROCEDURAL COMPONENT

The Java components of the system fall into four categories: Graphical User Interface (GUI), speech I/O, route finding, and scene interaction routines.

The GUI is built using the older Java AWT (Abstract Windowing Toolkit) rather than the newer Swing Set toolkit since Microsoft JVM 1.1 is a pre-Swing Java release. The GUI is illustrated in Figure 3 and provides for instant transport to any compartment in the model, escort between any two compartments, the ability to show or hide a trail of arrows delineating the path from an origin to a destination, and identification of the current compartment (including level, compartment number and name) and direction of view (forward, aft, port or starboard). A button is also provided to enable and disable speech recognition so the operator can carry on side conversations.

Fig. 3: Interactive Ship Familiarization System graphical user interface

The speech interface is written using the Java Speech API (JSAPI), a high-level programmer interface to JSAPI-compliant speech systems such as (in the current implementation) IBM ViaVoice. A speech command grammar written in Java Speech Grammar Format (JSGF) associates tags with subcomponents of the recognized utterance, reducing it to a simplified form that is more easily handled by the speech command interpreter. Spoken commands and queries generally correspond one-to-one to the controls in the GUI and update them when executed, e.g. the query *Where am I?* not only elicits a synthetic speech response identifying the current space ("This is the Pilot House") but also updates the value of the "Here" (current space) field in the graphical interface. The only difference from the GUI is that the speech interface provides separate queries for current level (*Which level am I on?*) and compartment number (*What's the number of this compartment?*), whereas in the GUI the number (which begins with the level) is always displayed alongside the name so as to identify it unambiguously, e.g. "08-162-1-L

Officer WC" and “09-163-1-L Officer WC". The speech synthesizer pronounces compartment numbers in Navy fashion (e.g. “oh eight tack one sixty two tack one tack L”) and the speech recognizer is prepared to recognize numbers read in the same way. Finally, the command *Stop listening* puts the speech recognizer into a “sleep” mode in which it ignores all further microphone input other than the command *Resume listening*, which restores it to full command recognition mode.

The route finder is a direct port to Java of the C language version used in the Shadwell project. It loads a waypoint connectivity file that defines the adjacencies of key coordinate locations (waypoints) in the model, such as doorways, corners (turn points) and the centers of compartments. When a route is requested, a separate resource file is first consulted to map origin and destination compartment numbers to their key waypoints, and then a best-first search finds the most cost-efficient route between the two points. A route can have a high cost either because of its overall length, or because it includes one or more waypoints that have been defined as being less preferable to use, e.g. a hatchway that requires stooping and crawling through, or doorways that take one from inside the ship to outside and vice versa. Once a route has been computed, the system either draws a chain of arrows along the path for the user to navigate (GUI command **Show Path** or verbal command *How do I get to…)*, or escorts the user along the path at a walking pace (GUI command **Escort** or verbal command *Take me to…*).

The scene interaction routines consist of escort (walkthrough) animation, route display, teleportation, viewpoint queries, and database queries, described next.

Escort Animation

The escort animation code takes the path (sequence of waypoint coordinates) generated by the route finder and converts it to a VRML PositionInterpolator node. As a clock signal is routed to the node, a continuous stream of interpolated coordinates is generated and routed to the **position** field of a dedicated Viewpoint node to which the user’s viewpoint has been temporarily bound, thus producing the animation. The clock is set and the interpolation generated in such a way that each clock tick results in an equal coordinate displacement, producing a steady transition at a walking pace. The route path is also used to generate an OrientationInterpolator node that is routed to the viewpoint’s **orientation** field to control the viewpoint’s horizontal (left-right) viewing angle. Since the orientation at each waypoint of the path is in the direction of the next waypoint, the OrientationInterpolator produces a continuous stream of interpolated rotations as if the user’s head is gradually turning in the direction s/he is about to move, rather than always looking straight ahead even at turns [as was the case in the Shadwell MSFT?]. The view remains level unless the user is ascending or descending, in which case the orientation is multiplied by a ±22.5° vertical vector causing the viewpoint to momentarily peer up or down and then level out again as the ascent/descent is completed. (Since this SFRotation.multiply() method is not provided in VRML Java Platform Scripting but only in the ECMAScript interface, it is performed by a separate Javascript routine.) While many people might just look straight ahead when using stairs or a ladder, the vertical orientation dipping is intended to give the user a preview of the space about to be entered so as to enhance their spatial awareness of the transition.

If the path’s origin waypoint corresponds to the same space the user is currently in (e.g. *Take me from **here** to…*), the user’s orientation is first rotated toward that waypoint. If the user is located more than a half meter from the waypoint s/he is then walked to its coordinates, at which point the path traversal itself begins. One problem with this approach is that even if the user’s current location is closer to a subsequent node on the path than to its origin, the user is still escorted to the origin and then has to “double back” past their original position. The straight-line walk to the

path origin is also incapable of navigating around obstacles like furniture or equipment, and in an irregularly shaped space might even go through an intervening wall. The solution to these problems would require predefining many more waypoints per space and modifying the route finder to associate multiple waypoints with each one, finding a path that originates from the closest such waypoint to the user.

If the user requests to be escorted from some other space to a destination, s/he is first teleported to the origin waypoint in that space (facing in the same direction as originally) and after a one second pause – intended to provide time to adjust to the "shock" of teleportation – is rotated in the direction of the path and the escort is begun. It might be argued that the user's orientation after the teleportation (or at the end of an escort, for that matter) should instead be in some pre-determined direction that provides a "good" view of the space, since under the current approach the user might arrive simply facing a blank wall. In the Shadwell system each waypoint was associated with a predefined orientation (an approach we also took in the ITD Kiosk project, since there the origin and destination points were typically hallway office doors with nameplates), and that approach could also be adopted here. The advantage to the current approach, however, is that if the user is aware of which direction they are facing before the teleportation, they can take advantage of that information afterwards without having to make an additional orientation query (*Which way am I facing now?*).

When an escort has been completed the system signals the arrival with synthesized speech ("You have arrived at the destination"). The Shadwell system also provided an additional visual cue (a large floating red ball) indicating the end of the path, but the current implementation relies just on the speech cue and cessation of motion (and/or end of arrow trail, see next section) to signal arrival at the destination. Finally, at any time during an escort the user can halt the walkthrough (e.g. *Stop the escort*) to pursue independent navigation or to request a subsequent escort or route display.

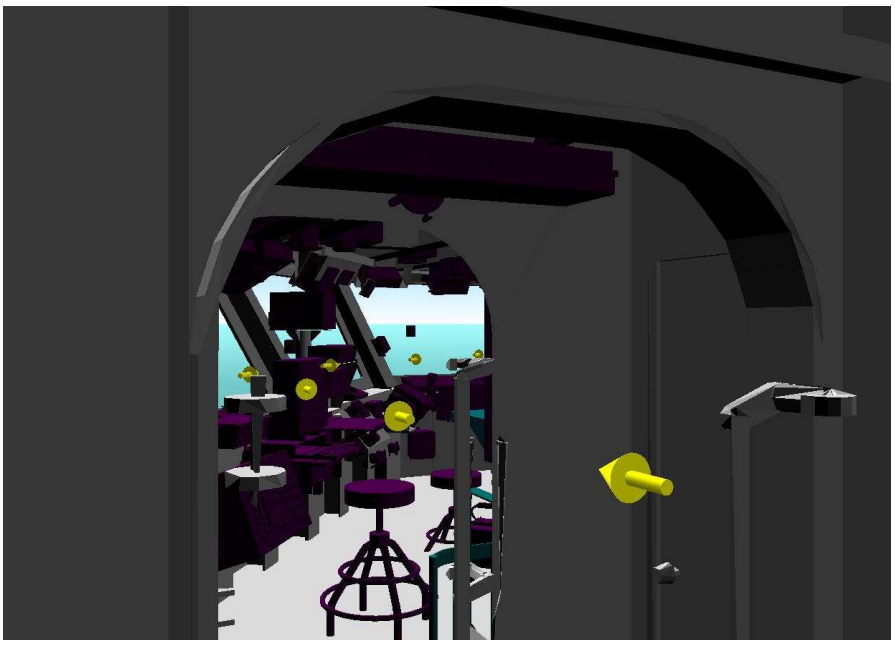

Fig. 4: Arrows show path computed by route finder

Route Display

The user can also request to be shown a path from one compartment to another, and then navigate the path manually using the computer's mouse and/or arrow keys. The path is shown as a sequence of bright yellow 3D arrows floating at about chest level and spaced 1.5 meters apart, each arrow pointing directly toward the one in front of it (Figure 4). (Since from the browser's viewpoint the arrows are virtual physical objects just like any other, they are grouped in a Collision node with collision disabled so they will not block the user from navigating the path if collision detection has been turned on.) As with virtual escort, if the origin waypoint of the path does not correspond to the space where the user is currently located, the user is first teleported to that waypoint. Instead of rotating the user's viewpoint toward the first arrow of the path, however, synthesized speech instead announces "Follow the arrows *<direction>*" where *<direction>* is "to your left/right", "in front of you" or "behind you". This gives control over any initial reorientation to the user, who in requesting a marked route most likely intends to navigate it manually rather than be given an automatic escort. It should also be noted that in a large irregularly shaped space like the Pilot House the initial arrow of the path might still not be visible even after a viewpoint rotation, in which case the user would have to do some navigation just to locate the start of the path. We experimented with adding arrows from the user's current position to the path origin, but this created unnecessarily confusing paths in the case of the "doubling back" routes described earlier.

If the user subsequently issues an escort command, s/he will be led along exactly the same sequence of positions as the arrow path, although during turns the arrows may momentarily go out of sight because the viewpoint orientation is being gradually interpolated away from the current straight-line path segment toward the direction of the next path segment. The user can also request at any time that the arrows be hidden or redisplayed. In the current implementation the arrow path remains visible even after the user has navigated or been escorted to the destination, in case they wish to retrace their steps.

Teleportation

Rather than be escorted at a walking pace to a new location, the user can request to be transported there instantly using commands such as *Teleport (beam, transport) me to Fly Control*. This is identical to the instant transport that occurs at the beginning of an escort or path display originating at a location other than the user's current location.

Viewpoint Queries

The user can at any time (including during an escort) query the name of the compartment they are currently in, the ship level they are on, or the direction (forward, aft, port, starboard) they are currently facing. Queries can be posed either as WH-questions (*What compartment is this? Which way am I facing? Which direction is starboard?*) or yes/no questions (*Is this the Pilot House? Am I facing aft? Is starboard to my right?*) The system determines the current compartment from the viewpoint coordinates by consulting a separately defined compartment perimeter database, just as in the Shadwell project. The ship level is determined from the viewpoint's vertical coordinate, and viewing direction from the viewpoint's orientation field. It should be noted that the current compartment and direction fields of the GUI do not automatically update as the user navigates or is escorted around, but only when explicitly queried, keeping the semantics of the GUI aligned with the semantics of the speech interface which similarly only provides such information when asked.

Database Queries

The user can query the compartment name, number and ship level for any particular space either by naming it (*What's the number of the Pilot House?*), giving its number (*What's the name of compartment oh eight tack one sixty eight tack three tack L?*), using an anaphoric reference (*How do I get to the Pilot House? What is* ***that compartment's*** *number? What level is* ***it*** *on?*), or a reference to the current space (*What's the number of this compartment?*). Most such queries can also be couched as yes/no questions, e.g. *Is the Pilot House on level 08*? If the name reference is ambiguous, the system chooses the one on the user's current level and responds accordingly:

What's the number of the Passage?	The one on this level is number 08-168-1-L.
What level is the port weather deck on?	There is one on this level, 09.
Is the forward weather deck on Level 08?	There is one on that level.

SPEECH COMMAND TAGGING

The following examples show the consistent manner in which path-based requests are tagged in the speech recognition grammar. When unspecified the origin is assumed to be the current compartment and is tagged **here**, the same as the explicit *here, this compartment,* etc. The origin and destination are additionally tagged by the keywords **from** and **to** since syntactically they can occur in either order. The command interpreter dereferences the token **there** (*there, that compartment, it,* etc.) as the most recently mentioned space from a prior sentence.

How do I get from Fly Control to the Pilot House?	show from 09-160-1-C to 08-159-1-C
Show me how to get from here to the Pilot House.	show from here to 08-159-1-C
Where is the Pilot House?	show from here to 08-159-1-C
Show me where the Pilot House is from Fly Control.	show to 08-159-1-C from 09-160-1-C
Show me how to get there.	show from here to there

Take me from Fly Control to the Pilot House.	escort from 09-160-1-C to 08-159-1-C
Show me from here to the Pilot House.	escort from here to 08-159-1-C
Escort me to the Pilot House.	escort from here to 08-159-1-C
Walk me to the Pilot House from Fly Control.	escort to 08-159-1-C from 09-160-1-C
Take me there.	escort from here to there

Note that *Where is <name>?* is interpreted in this implementation as a request to be shown a path to the named space, unlike the earlier Shadwell interface in which it was interpreted as a request for its compartment number. An ambiguous name reference produces a tag structure containing an embedded list of alternative compartment number strings:

How to I get to the Officer WC?	show from here to {08-162-1-L 09-163-1-L}
Escort me from the Bath to this compartment.	escort from {08-168-3-L 09-167-1-L} to here

If the ambiguous reference is the destination, the command interpreter chooses the one on the same level as the origin as being the most likely intended goal; if the origin, it chooses the one on the level the user is currently on. In either case it announces "There is more than one <name>, going to the one on the [current, same] level." Since these heuristics are not guaranteed to yield what the user intended, a better approach might be to engage the user in a dialogue to resolve any ambiguities. To avoid any confusion to begin with the user can also preface the name with its level, e.g. *How do I get to the 09 Level Officer WC?*

Since the route finder only associates one key waypoint with each compartment, referencing the exterior weather decks becomes problematic because they wrap around the entire Island House and we would like a command like *How do I get to the weather deck?* to find a path to the nearest segment of the deck. For that reason we retain the approach taken in MSFT by treating the port, forward and starboard weather decks on each level as different "compartments" and requiring that the user reference them specifically (*Take me to the forward weather deck*) subject to the same level disambiguation routine just described.

FUTURE WORK

While VRML is an object-oriented representation in which shapes can be hierarchically grouped into named nodes representing individual objects, those internal names may not be adequate to identify entities for reference by a spoken language interface. For example, one of the deck chairs in Fly Control is represented by a VRML Transform node named **_76_5033_1331**, but without a concordance between an equipment list and such identifiers it would be impossible to implement referential capability such as *that chair, the nearest fire extinguisher*, etc. in the interface.

REFERENCES

Tate, David L., Stephanie S. Everett, Tucker Maney, and Kenneth Wauchope (2000). A Multimodal Virtual Environment for Ship Familiarization. NRL/FR-MM/6180--00-9939, Naval Research Laboratory, Washington, DC.